Continuation Passing Style

We introduced the CK-Machine as a model of an interpreter that does not require a stack. Instead, it explicitly represents the continuation as a data structure. We are now going to achieve a same result, but in a very different way: (1) we will to identify a subset of expressions that do not use the the stack and (2) we will compile arbitrary expressions to this subset in a systematic way. The advantage of this approach is that it lets us reuse our existing interpreter and we do not need to implement an entirely new CK-Machine.

Figure 29.1a shows the syntax of language that we will work with. Note that programs in this language *do* require the stack. This language has arithmetic, conditionals, let-expressions, the c expression, and n-argument functions. We do not strictly need n-argument functions, but they will make our work a lot easier. Figure 29.1b is the subset of of expressions that do not require the stack. The key idea is that we stratify syntax of the language into atomic expressions (a) that obviously do not require the stack and complex expressions (e) that only use the stack in a trivial way.

For example, notice that the arithmetic expression (30 + 4) + (1000 + 200) is a valid expression in the original syntax, but is not a valid expression in the subset. In fact, we need a stack to evaluate it. Using the CK Machine, we can see that the stack grows to a depth of 2:

$$
\begin{array}{rl}
 & (30 + 4) + (1000 + 200),\ [] \\
\rightarrow & 30 + 4,\ [\mathbf{add}_R(1000 + 200)] \\
\rightarrow & 30,\ [\mathbf{add}_R(4),\ \mathbf{add}_R(1000 + 200)] \\
\rightarrow & 4,\ [\mathbf{add}_L(30),\ \mathbf{add}_R(1000 + 200)] \\
\rightarrow & 34,\ [\mathbf{add}_R(1000 + 200)] \\
\rightarrow & 1000 + 200,\ [\mathbf{add}_L(34)] \\
\rightarrow & 1000,\ [\mathbf{add}_R(200),\ \mathbf{add}_L(34)] \\
\rightarrow & 200,\ [\mathbf{add}_L(1000),\ \mathbf{add}_L(34)] \\
\rightarrow & 1200,\ [\mathbf{add}_L(34)] \\
\rightarrow & 1234,\ []
\end{array}
$$

In general, an arithmetic expression that is n-level deep will require a stack of depth n. We can rewrite this expression to an equivalent expression that is in the subset, e.g., **let** $x = 30 + 4$ **in let** $y = 1000 + 200$ **in** $x + y$, which we evaluate as follows:

$$
\begin{array}{rl}
 & \textbf{let } x = 30 + 4 \textbf{ in let } y = 1000 + 200 \textbf{ in } x + y,\ [] \\
\rightarrow & \textbf{let } y = 1000 + 200 \textbf{ in } 34 + y,\ [] \\
\rightarrow & 34 + 1200,\ [] \\
\rightarrow & 1234,\ []
\end{array}
$$

Note. If you carefully derive the above reduction sequence using the CK Machine, you'll find that it uses the stack. To address this problem, we can add these two rules to our CK Machine:

add and substitute	**let** $x = m + n$ **in** $e, \vec{\kappa} \rightarrow e[x/r], \vec{\kappa}$ where $r = m + n$
add immediate	$m + n, \vec{\kappa} \rightarrow r, \vec{\kappa}$ where $r = m + n$

With these two rules, we can derive the reduction sequence correctly.

Similarly, notice the subset requires all applications to be of the form $a(a_1, ..., a_n)$. This means we cannot have nested function applications, e.g., $f(g(x))$ is not in the subset. Applications cannot appear in arithmetic expressions either, e.g., $f(x) + 1$ is not in the subset. In fact, we cannot name the result of an application, e.g., **let** $x = f(y)$ **in** e is not in the subset either. To summarize, this subset requires *all applications to be in tail position*, which we should expect, because tail calls do not consume stack space.

e	=	n	numbers
	|	b	booleans
	|	x	identifiers
	|	$e_1 + e_2$	addition
	|	**fun** x_1, ..., x_n . e	n-argument function
	|	$e(e_1, ..., e_n)$	application
	|	**let** $x = e_1$ **in** e_2	let-expression
	|	**if** e_1 **then** e_2 **else** e_3	conditional
	|	$\mathcal{C}(e)$	continuation capture

(a) Original syntax.

a	=	n	numbers
	|	b	booleans
	|	x	identifiers
	|	**fun** x_1, ..., x_n . e	n-argument function
e	=	a	atomic expression
	|	**let** $x = a$ **in** e	named atomic expression
	|	**let** $x = a_1 + a_2$ **in** e	named addition
	|	$a(a_1, ..., a_n)$	application
	|	**if** a_1 **then** e_2 **else** e_3	conditional

(b) CPS syntax.

NUM	$\mathsf{cps}(n) = \textbf{fun}\ k\ .\ k(n)$
BOOL	$\mathsf{cps}(b) = \textbf{fun}\ k\ .\ k(b)$
ID	$\mathsf{cps}(x) = \textbf{fun}\ k\ .\ k(x)$
ADD	$\mathsf{cps}(e_1 + e_2) = \textbf{fun}\ k\ .\ \mathsf{cps}(e_1)(\textbf{fun}\ x\ .\ \mathsf{cps}(e_2)(\textbf{fun}\ y\ .\ \textbf{let}\ r = x + y\ \textbf{in}\ k(r)))$
FUN	$\mathsf{cps}(\textbf{fun}\ x_1, ..., x_n\ .\ e) = \textbf{fun}\ k\ .\ k\ (\textbf{fun}\ k', x_1, ..., x_n\ .\ \mathsf{cps(e)}\ \mathsf{k'})$
APP	$e_f(e_1, ..., e_n) = \textbf{fun}\ k\ .\ \mathsf{cps}(e_f)(\textbf{fun}\ v_f\ .\ \mathsf{cps}(e_1)(\textbf{fun}\ v_1\ .\ ...\ \mathsf{cps}(e_n)(\textbf{fun}\ v_n\ .v_f(k,v_1, ...,v_n))))$
LET	$\mathsf{cps}(\textbf{let}\ x = e_1\ \textbf{in}\ e_2) = \textbf{fun}\ k\ .\ \mathsf{cps}(e_1)(\textbf{fun}\ v1\ .\ \textbf{let}\ x = v1\ \textbf{in}\ \mathsf{cps}(e_2)\ k$
IF	$\mathsf{cps}(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) = \textbf{fun}\ k\ .\ \mathsf{cps}(e_1)(\textbf{fun}\ v1\ .\ \textbf{if}\ v1\ \textbf{then}\ \mathsf{cps}(e_2)(k)\ \textbf{else}\ \mathsf{cps}(e_2)(k))$
$\mathcal{C}$	$\mathsf{cps}(\mathcal{C}(e)) = \textbf{fun}\ k\ .\ \mathsf{cps}(e)(\textbf{fun}\ f\ .\ f\ (\textbf{fun}\ x\ .\ x, \textbf{fun}\ (k', \mathsf{v})\ .\ \mathsf{k(v)}))$

(c) Naive CPS conversion.

Figure 29.1: Syntax of source

Note. This is not quite true for n-argument functions, and there are two ways to resolve this. The simplest approach is to add the following reduction rule:

$$(\textbf{fun}\ x_1, ..., x_n\ .\ e)(a_1, ..., a_n), \vec{\kappa} \to e[x_1/a_1, ..., x_n/a_n], \vec{\kappa}$$

This rule exploits the observation that applications in the subset do not have complex subexpressions, thus we can substitute immediately. Alternatively, we can leave our definitions unchanged, in which case a program in the subset will consume at most N stack frames, where N is the maximum number of arguments that any function receives.

The key restriction in this subset is that all applications are in tail position, thus we cannot have a one function return its result to another using the stack. For example, the following program is not allowed:

let g = **fun** x . $x + 1$ **in**
$f(g(20))$

However, we can rewrite g to take f as an argument and make a (tail) call to the argument with its result:

let g = **fun** $k\ x$. **let** $r = x + 1$ **in** $k(r)$ **in**
$g(f, 20)$

These two programs are equivalent and the latter is in the subset.

The key idea of our transformation is to rewrite all functions f to take an additional argument k, where k is itself a function that represents the continuation of f. We can generalize this idea to all expressions: we will turn all expressions e into functions **fun** k . e' such that e' (tail) calls k with the value of e. This is known as *continuation passing style* (CPS).

1 Naive CPS Transformation

We now define a function $\mathsf{cps} : e \to e$ that transforms programs in the original syntax to programs in continuation passing style. The function that we present in this section is not efficient. We will address this problem in the next section. The complete cps function is in fig. 29.1c. We will examine it small pieces.

Arithmetic The ADD and NUM cases all that are necessary to CPS simple arithmetic expressions. These two rules can produce enormous expressions, e.g.:

$$\begin{array}{ll} & \mathsf{cps}(1 + 20) \\ = & \textbf{fun } k \text{ . } \mathsf{cps}(1)(\textbf{fun } x \text{ . } \mathsf{cps}(20)(\textbf{fun } y \text{ . } \textbf{let } r = x + y \textbf{ in } k(r))) \\ = & \textbf{fun } k \text{ . } (\textbf{fun } k2 \text{ . } k2(1))(\textbf{fun } x \text{ . } \mathsf{cps}(20)(\textbf{fun } y \text{ . } \textbf{let } r = x + y \textbf{ in } k(r))) \\ = & \textbf{fun } k \text{ . } (\textbf{fun } k2 \text{ . } k2(1))(\textbf{fun } x \text{ . } (\textbf{fun } k3 \text{ . } k3(20)))(\textbf{fun } y \text{ . } \textbf{let } r = x + y \textbf{ in } k(r))) \end{array}$$

However, we can safely apply the generated functions to simplify the expression further:

$$\begin{array}{lll} & \textbf{fun } k \text{ . } (\textbf{fun } k2 \text{ . } k2(1))(\textbf{fun } x \text{ . } (\textbf{fun } k3 \text{ . } k3(20)))(\textbf{fun } y \text{ . } \textbf{let } r = x + y \textbf{ in } k(r))) & \\ = & \textbf{fun } k \text{ . } (\textbf{fun } k3 \text{ . } k3(20))(\textbf{fun } y \text{ . } \textbf{let } r = 1 + y \textbf{ in } k(r)) & \text{apply } \textbf{fun } k2 \text{ . } k2(1) \text{ to its argument} \\ = & \textbf{fun } k \text{ . } \textbf{let } r = 1 + 20 \textbf{ in } k(r)) & \text{apply } \textbf{fun } k3 \text{ . } k3(1) \text{ to its argument} \end{array}$$

The expression above is a function and not an arithmetic expression. However, we can apply this function to the identity function to get the expected arithmetic expression:

$$\begin{array}{ll} & (\textbf{fun } k \text{ . } \textbf{let } r = 1 + 20 \textbf{ in } k(r))\ (\textbf{fun } x \text{ . } x) \\ \rightarrow & \textbf{let } r = 1 + 20 \textbf{ in } (\textbf{fun } x \text{ . } x)(r) \\ = & \textbf{let } r = 1 + 20 \textbf{ in } r \\ = & 1 + 20 \end{array}$$

Correctness For any expression e, $\mathsf{cps}(e)$ produces a function, which we can apply to the identity function to get the same result as e. We have about what we mean by "same" result. Notice that $\mathsf{cps}(n)(\textbf{fun } x \text{ . } x) = n$. In fact, if e produces a number n then $\mathsf{cps}(e)(\textbf{fun } x \text{ . } x)$ will also produce n. However, if e produces a function f then CPSing will produce the CPSed version of f.

Theorem 1 (Correctness of CPS)**.** *For all expressions e, $e,[] \rightarrow* v,[]$ if and only if $\mathsf{cps}(e)(\textbf{fun } x \text{ . } x), [] \rightarrow* \mathsf{cps}(v)(\textbf{fun } x \text{ . } x), []$*

Functions and applications The FUN case gives every function an extra argument and CPSes its body. For example:

$$\begin{array}{ll} & \mathsf{cps}(\textbf{fun } x \text{ . } x + 1) \\ = & \textbf{fun } k \text{ . } k\ (\textbf{fun } k',\mathsf{x} \text{ . } \mathsf{cps}(\mathsf{x} + 1)\ \mathsf{k}') \\ = & \cdots \\ = & \textbf{fun } k \text{ . } k\ (\textbf{fun } k',\mathsf{x} \text{ . } (\mathsf{fun}\ \mathsf{k} \text{ . } \mathsf{let}\ \mathsf{r} = \mathsf{x} + 1\ \mathsf{in}\ \mathsf{k2}(\mathsf{r}))\ \mathsf{k}') \\ = & \textbf{fun } k \text{ . } k\ (\textbf{fun } k',\mathsf{x} \text{ . } \mathsf{let}\ \mathsf{r} = \mathsf{x} + 1\ \mathsf{in}\ \mathsf{k}'(r)) \end{array}$$

If we apply this function to the identity function, we get $\textbf{fun } k',\mathsf{x} \text{ . } \mathsf{let}\ \mathsf{r} = \mathsf{x} + 1\ \mathsf{in}\ \mathsf{k}'(r)$. This function computes $x + 1$—like the original function—but passes the result to k instead of returning the result.

The APP case transforms every application to pass its continuation to the function:

$$\begin{array}{ll} & \mathsf{cps}(f(20)) \\ = & \textbf{fun } k \text{ . } \mathsf{cps}(f)(\textbf{fun } v_f \text{ . } \mathsf{cps}(20)\ (\textbf{fun } v \text{ .} v_f(k,v))) \\ = & \cdots \\ = & \textbf{fun } k \text{ . } f(k,20) \end{array}$$

The $\mathcal{C}$ operator You may have noticed that the source language includes the C operator and that the target language does not. This is not an error of omission. It turns out that we can translate $\mathcal{C}$ to an ordinary function call when the program is in continuation-passing style. The case for C is very short:

$$\mathsf{cps}(\mathcal{C}(e)) = \textbf{fun } k \text{ . } \mathsf{cps}(e)(\textbf{fun } f \text{ . } f\ (\textbf{fun } x \text{ . } x,\ k))$$

To understand why this works, let's consider the special case where the argument to C is a literal function (recall that the argument must evaluate to a function):

$$\begin{array}{ll} & \mathsf{cps}(\mathcal{C}(\textbf{fun } x \text{ . } e)) \\ = & \textbf{fun } k \text{ . } \mathsf{cps}(\textbf{fun } x \text{ . } e)(\textbf{fun } f \text{ . } f\ (\textbf{fun } x \text{ . } x,\ k)) \\ = & \textbf{fun } k \text{ . } (\textbf{fun } k_2 \text{ . } k_2\ (\textbf{fun } k_3,\ x \text{ . } \mathsf{cps}(e)(k_3)))\ (\textbf{fun } f \text{ . } f\ (\textbf{fun } x \text{ . } x,\ k)) \\ = & \textbf{fun } k \text{ . } (\textbf{fun } f \text{ . } f\ (\textbf{fun } x \text{ . } x,\ k))\ (\textbf{fun } k_3,\ x \text{ . } \mathsf{cps}(e)(k_3)) \\ = & \textbf{fun } k \text{ . } (\textbf{fun } k_3,\ x \text{ . } \mathsf{cps}(e)(k_3))\ (\textbf{fun } x \text{ . } x,\ k) \\ = & \textbf{fun } k \text{ . } \mathsf{cps}(e[x/k])(\textbf{fun } x \text{ . } x) \end{array}$$

```
type id = string
type exp =
  | Num of int
  | Id of id
  | Add of exp * exp
  | Let of id * exp * exp
  | Fun of id * exp
  | App of exp * exp
```

(a) Syntax.

```
let rec cps  (e : exp) : exp = match e with
  | Num n -> Fun ("k", App (Id "k", Num n))
  | Add (e1, e2) -> Fun ("k",
      App (cps e1) Fun ("v1",
        App (cps e2) Fun ("v2",
          Let ("r", Add (Id "v1", Id "v2"),
            App (Id "k", Id "r")))))
```

(b) Naive CPS.

```
let rec cps (e : exp) : exp = match e with
  | Num n -> fun k -> k (Num n)
  | Add (e1, e2) -> fun k ->
      (cps e1) (fun (v1 : exp) ->
        (cps e2) (fun (v2 : exp) ->
          Let ("r", Add (Id "v1", Id "v2"),
            k (Id "r"))))
```

(c) Efficient CPS.

Figure 29.2: Two ways to convert arithmetic expressions to CPS.

In the simplified expression above, we can see that the body e receives the identity function as its continuation and that the actual continuation k is bound to x. Therefore, when e returns normally it calls the identify function and when e applies x it restores the original continuation. This is exactly the same behavior we got from the CK-machine.

2 A Better CPS Transformation

The CPS function from the previous section produces a lot of extra functions and function applications at every step that clutter the output. (These are known as *administrative functions* and *administrative applications.*). It usually helps to apply administrative functions to make the output more compact and readable. In fact, we can avoid producing many administrative functions altogether.

It will be easier to present this new algorithm in code. Figure 29.2b implements the NUM and ADD rules from fig. 29.1c. Figure 29.2c shows the efficient algorithm that avoids generating administrative code. The efficient algorithm differs from the naive algorithm in two ways: (1) we replace administrative functions (`Fun ("k",`) with OCaml functions (`fun k ->`) and (2) we replace the administrative application (`App (Id "k", Id "r")`) with an OCaml application (`k (Id "r")`). Therefore, the administrative functions do exist, but since we turn them into OCaml functions, they get applied in the process of compilation.

Finally, recall that the naive algorithm produces a function in the object language. Therefore, we have to finish by applying this function to the identity function in the object language:

`App (cps` e`, Fun ("x", Id "x"))`

However, the efficient algorithm produces a function *in OCaml*. Therefore, we finish by applying this function to the identity function in OCaml:

`cps` e `(fun x -> x)`

Figure 29.3 is an implementation of the efficient CPS algorithm for the entire language, including n-argument functions, conditionals, and the `control` operator.

```
type id = string

let fresh_id : unit -> id =
  let n = ref 0 in
  fun () ->
    let x = "tmp" ^ string_of_int !n in
    n := !n + 1;
    x

type exp =
  | Num of int
  | Bool of bool
  | Id of id
  | Add of exp * exp
  | Fun of id list * exp
  | App of exp * exp list
  | Let of id * exp * exp
  | If of exp * exp * exp
  | Control of exp


let rec cps (exp : exp) : (exp -> exp) -> exp = match exp with
  | Num n -> fun k -> k (Num n)
  | Bool b -> fun k -> k (Bool b)
  | Id x -> fun k -> k (Id x)
  | Add (e1, e2) -> fun k ->
    cps e1 (fun x1 ->
      cps e2 (fun x2 ->
        let r = fresh_id () in
        Let (r, Add (x1, x2), k (Id r))))
  | Let (x, e1, e2) -> fun k ->
    cps e1 (fun v1 ->
      Let (x, v1, cps e2 k))
  | Fun (xs, e) -> fun k ->
    let k' = fresh_id () in
      k (Fun (k' :: xs,
         cps e (fun r -> App (Id k', [r])))))
  | App (f, args) -> fun k ->
    cps f (fun f_v ->
      cps_args args (fun args_v ->
        App (f_v, k :: args_v)))
  | If (e1, e2, e3) -> fun k ->
    let k' = fresh_id () in
    let k'_f = fun r -> App (Id k', [r]) in
    Let (k', (let r = fresh_id () in Fun ([r], k (Id r))),
      cps e1 (fun v1 ->
        If (v1, cps e2 k'_f, cps e3 k'_f)))
  | Control (e) -> fun k ->
      cps e (fun f -> App (f, [Fun (["x"], Id "x");
      Fun (["k"; "x"], k (Id "x"))]))

and cps_args (args : exp list) (k : exp list -> exp) = match args with
  | [] -> k []
  | arg :: args' -> cps arg (fun a_v ->
      cps_args args' (fun args'_v ->
        k (a_v :: args'_v)))
```

Figure 29.3: An efficient CPS algorithm.